Guida al linguaggio GML di Game Maker

Programma e Programmazione

Iniziamo dall'affermazione più banale in assoluto: un **videogioco** non è altro che un **programma**. A questo punto, non rimane altro da fare che spiegare cosa è un programma! Un programma non è altro che la descrizione di un problema da risolvere, scritto in modo tale da diventare comprensibile ad un computer.

Nel nostro caso, il problema da risolvere è il videogioco, composto di immagini e suoni che interagiscono, tra loro e con l'utente, secondo un dato schema logico.

La **programmazione**, è il processo di realizzazione di un programma usando un **linguaggio** di programmazione per rendere il nostro problema comprensibile al calcolatore.

Questo processo si può idealmente dividere in due fasi: prima si risolvono i problemi a livello logico, poi si traduce la soluzione nel linguaggio di programmazione scelto, usando gli strumenti che ci mette a disposizione.

Se nella prima fase si studia il risultato che ci si aspetta, senza particolari restrizioni, nella seconda fase è importante rispettare le regole che ogni **linguaggio** impone.

Cosa è GML

Il GML è un linguaggio interpretato di programmazione strutturata.
La teoria della programmazione strutturata dimostra che qualsiasi programma si può scrivere usando solo tre tipi di strutture di controllo:

- la sequenza, in cui le istruzioni sono semplicemente accodate le une alle altre.
- l'iterazione, in cui le istruzioni vengono ripetute ciclicamente.
- l'alternativa, in cui le istruzioni vengono eseguite solo a certe condizioni.

Il **GML** è simile al **C**: hanno in comune gran parte delle strutture di controllo e buona parte della sintassi.

Introduzione al Game Maker Language (GML) - parte I

Il sistema **DnD** di Game Maker è semplice da usare, ma è poco indicato alla realizzazione di oggetti complessi.
Ad un oggetto possono essere affidati molti comportamenti che siano facilmente modificabili ed espandibili. E' quindi più **pratico** abbinare agli eventi, cui l'oggetto deve rispondere, un pezzo di codice che descriva tutte le azioni che l'oggetto deve compiere, anzichè la loro rappresentazione grafica. Andiamo ora a vedere come e dove si può inserire il codice e quali sono le strutture di controllo che possiamo usare.

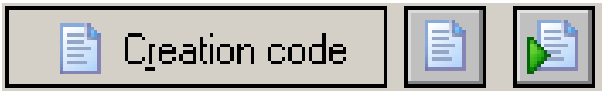
Come e dove inserire il codice

In **Game Maker** esistono due punti in cui è possibile inserire del codice: all'interno di un oggetto, in risposta ad un evento, nella colonna delle action; nella room, come 'creation code'.

Per inserire **codice** in un oggetto, come risposta ad un evento, è possibile trascinare l'icona 'Execute a Piece of Code' (presente nel tab 'control') direttamente nella colonna delle action. Questa operazione aprirà lo **script editor** in cui è possibile scrivere il nostro codice.

L'icona a fianco ad 'Execute a Piece of Code', porta il nome di 'Execute a Script'. E' possibile trascinarla nella colonna delle action, ma questa non aprirà direttamente lo script editor. Richiederà invece di scegliere tra degli script preventivamente scritti. Per rendere uno script disponibile per questa operazione, basta aggiungerlo tra le risorse di gioco con la voce 'Add Script' del menù Add (resource in GM7).

Il 'Creation Code' verrà eseguito alla creazione della **room** che lo contiene. Per poterlo scrivere, andare nelle *room properties*, selezionare il tab *setting* e premere il pulsante 'Creation code'.



Creation code, Piece of Code, Script

Programma e strutture di controllo

Un programma in GML è essenzialmente una sequenza di istruzioni:

```
{
        ...
    < istruzione 1>;
    < istruzione 2>;
    < istruzione 3>;
        ...
}
```

Ci sono alcune istruzioni che vanno eseguite solo a determinate **condizioni**. E' quindi spesso necessario dover scegliere tra blocchi di istruzioni alternative. **GML** permette di attuare questa scelta tramite due strutture diverse: **if-else** e **switch**.

La struttura **if** (l'**else** è facoltativo), permette di eseguire un blocco solo se una condizione è vera:

```
if condizione=TRUE
  {
  < istruzione 1>;
  < istruzione 2>;
  < istruzione 3>;
  }
```

Nell'esempio qui sopra, nel caso la condizione non fosse vera (**TRUE** è una ***costante*** uguale ad 1), non verrà eseguita alcuna istruzione. Possiamo usare **else** qualora dovessimo eseguire delle istruzioni in caso la condizione fosse falsa:

```
if condizione=TRUE
  {
    < istruzione 1>;
  }
else
  {
    < istruzione 2>;
  }
```

La struttura **switch** corrisponde a quella chiamata **case-select** in altri linguaggi e permette di scegliere tra diverse alternative in base al valore di una variabile di controllo:

```
switch (variabile_di_controllo)
    {
    case 1:
      < istruzione 1>;
      < istruzione 2>;
    break;
    case 2:
      < istruzione 3>;
      < istruzione 4>;
    break;
   }
```

Nell'esempio qui sopra, con *variabile_di_controllo* uguale ad 1 verrà eseguito il primo set d'istruzioni, se è uguale a 2 il secondo blocco d'istruzioni, con altri valori non verrà eseguito alcun codice.

Iterazioni

Un'iterazione è la ripetizione ciclica di un blocco d'istruzioni. GML mette a disposizione queste strutture di iterazione: **repeat**, **while**, **do-until** e **for**. La struttura più semplice è **repeat**, che ripete un blocco di codice un numero predefinito di volte:

```
repeat (5)
  {
    < istruzione 1>;
    < istruzione 2>;
    < istruzione 3>;
  }
```

Discorso diverso per tutti gli altri cicli, che vengono eseguiti fino al raggiungimento di una condizione o finchè una condizione è vera. Per esempio, il ciclo **while**, esegue il blocco controllato finchè una condizione risulta essere vera:

```
while (condizione=TRUE)
  {
    < istruzione 1>;
    < istruzione 2>;
    < istruzione 3>;
  }
```

Il ciclo **do-until**, al contrario, esegue il ciclo finchè una condizione rimane vera:

```
do
 {
    < istruzione 1>;
    < istruzione 2>;
    < istruzione 3>;
 } until (condizione=TRUE);
```

La differenza con **while** è che il ciclo **do-until** viene sempre eseguito almeno una volta. Bisogna ricordarsi di prestare molta attenzione quando si usano questi cicli, perchè è facile creare cicli infiniti in cui il programma potrebbe non rispondere più al controllo. Vediamo infine il ciclo **for**:

```
for(dichiarazione1; condizione; dichiarazione2)
 {
    < istruzione 1>;
    < istruzione 2>;
    < istruzione 3>;
 }
```

Il ciclo **for** è un ciclo a iterazione predefinita o condizionata la prima dichiarazione inizializza il contatore del ciclo la condizione verifica il valore del contatore o di altra condizione la seconda dichiarazione modifica, incrementando o decrementando, il contatore. Vediamo un esempio:

```
for(i=0; i<10; i+=1)
 {
    < istruzione 1>;
    < istruzione 2>;
     .....
 }
```

Nell'esempio, la prima dichiarazione inizializza il valore della variabile **i** a 0. Il ciclo viene ripetuto finchè la condizione "i minore di 10" risulta vera. **i** aumenta di 1 ad ogni ciclo. Il ciclo verrà eseguito 10 volte, con **i** che assume valori da 0 a 9. Quando **i** raggiungerà 10 la condizione risulterà falsa ed il ciclo non verrà eseguito.

Queste sono le strutture di controllo che **GML** fornisce per 'guidare' il flusso del codice. Abbiamo appurato che il programma è una sequenza di istruzioni, ma la loro esecuzione è vincolata a dalle condizioni. Ma cosa sono queste istruzioni?

Un'istruzione, è l'**ordine** dato al computer di eseguire delle **operazioni** su dei valori presenti in memoria. Possono essere operazioni algebriche, o operazioni booleane od operazioni di comparazione.

Queste operazioni, verranno eseguite su valori **contenuti** in variabili od altre strutture dati.

Variabili e strutture dati

Le istruzioni servono di solito a manipolare dei dati. Spesso questi dati sono contenuti in **variabili**. In **GML** una variabile può contenere sia valori numerici che stringhe di testo. L'operazione di attribuire un certo contenuto ad una variabile si chiama **assegnazione** e si compie con quel simbolo grafico che siamo abituati a chiamare 'uguale':

```
{
//assegno alla variabile 'a' il valore numerico 10
a=10;
//assegno alla variabile 'b' una stringa di testo
b='testo assegnato alla variabile';
}
```

In molti linguaggi di programmazione l'assegnazione e la comparazione di valori si fanno in modo diverso. Ad esempio, si assegna un valore con '=' e si compara con '=='. In **GML** è possibile eseguire la comparazione usando indifferentemente '=', '==' o ':=', mentre per l'assegnazione '==' genera un errore (Assignment operator expected).

```
{
a=10;  //assegnazione valida
a:=10;  //assegnazione valida
a==10; //errore di assegnazione
}
```

Il **nome** della variabile può contenere caratteri alfanumerici ed il simbolo underscore ('_'), ma non può iniziare con un numero. La massima lunghezza ammessa per un nome di una variabile è di 64 caratteri.

Per assegnare una stringa di testo ad una **variabile**, questa deve essere scritta tra apici o tra virgolette ('apici', "virgolette"). Tra virgolette viene mantenuta parte della formattazione del testo, tra apici no.

Le variabili possono essere definite dall'utente, ma esistono anche variabili **predefinite** (built-in variables). Tra le variabili predefinite troviamo quelle di uso così comune che sarebbe fastidioso doverle definire ogni volta, come ad esempio quelle che contengono la posizione **x** ed **y** di un oggetto sullo schermo.

Nello **script editor**, le variabili predefinite vengono visualizzate in **BLUE**, a differenza delle funzioni che vengono visualizzate color **NAVY**.

Alcune variabili predefinite sono di sola lettura e non si possono modificare con la normale assegnazione.
Le variabili definite all'interno di un oggetto, sono locali: sono valide solo all'interno di quell'oggetto. Per accedervi da altri oggetti è necessario indicare il nome della variabile, preceduto dal nome dell'oggetto (o dall' **id** di un'istanza) separati da un punto:

```
{
  nome_oggetto.variabile_locale=20;
}
```

Per rendere una variabile **globale**, cioè valida in tutto il gioco/programma, è necessario dichiararla appartenete all'oggetto speciale global:

```
{
  global.variabile_globale=20;
}
```

A volte sono necessarie variabili valide solo all'interno del blocco in cui sono dichiarate. Questo si può ottenere grazie all'istruzione **var**:

```
var a;
a=10;
while (a>5)
   {
     x=a;
     a-=1;
   }
```

Nell'esempio qui sopra, la variabile **a** esiste ed è valida SOLO nel blocco in cui è definita. Quando dobbiamo memorizzare sequenze di dati, usare delle variabili può risultare poco pratico. Possiamo usare altre strutture dati, come ad esempio il vettore **o array**.

Un **array** va immaginato come una sequenza di celle, ciascuna delle quali equivale ad una **variabile**. In GML ciascuna cella dello stesso **array** può contenere indifferentemente stringhe o numeri.
Ogni **cella** di un array è identificata da un'indice. In GML l'indice deve essere un numero intero, compreso tra 0 e 32000. Per assegnare un dato ad una cella di un array si indicando il nome del vettore e l'indice della cella:

```
  {
    mio_array[0]=232;
    mio_array[1]=0;
    mio_array[2]=640;
  }
```

Se i valori contenuti in un **array** sono riconducibili ad una sequenza, per l'assegnazione si può far ricorso ad un ciclo:

```
for (i=0; i<100; i+=1)
  {
    posizioni[i]=20+(i*5);
  }
```

Quando è necessario **memorizzare** coppie di valori, possiamo servirci di array **bidimensionali** detti anche **matrici**

```
  for (i=1; i<room_width+1; i+=1)
 {
   for (j=1; j<room_height+1 ; j+=1){
     colori[i,j]=draw_getpixel(i,j);
   }
 }
```

L'esempio qui sopra 'legge' i colori dell'immagine visualizzata in una room e le memorizza in una **matrice**.

Quando anche l'array bidimensionale si dimostrasse insufficiente per i nostri scopi, **GML** mette a disposizione strutture dati più complesse ed evolute: **stacks**, **queues**, **priority queues**, **lists**, **maps** e **grids**.

Lo **stack** va immaginato come una pila, nel senso di cumulo, di informazioni. E' una struttura dati ad accesso sequenziale di tipo **LIFO (last in, first out)**, cioè il primo valore immesso sarà l'ultimo recuperabile.

Le **queues** sono l'esatto opposto: strutture dati ad accesso sequenziale di tipo **FIFO (first in, first out)**, cioè l'ultimo valore immesso sarà l'ultimo recuperabile.

Le **lists** sono strutture dati ad accesso casuale, cioè i valori vengono inseriti secondo l'ordine voluto e recuperati alla stessa maniera.

Una **map** è strutturalmente abbastanza simile ad un array, ma fornisce alcune funzioni per cercare dei valori al suo interno.

Le **priority queues**, sono come le **queues**, ma il contenuto è organizzato per priorità d'accesso piuttosto che per ordine d'immissione.

Le **grids** sono la struttura dati più complessa fornita da **GML**. Ha una struttura simile all'array bidimensionale, ma fornisce diverse funzioni d'interrogazione dei contenuti.

Operatori logici e operatori aritmetici

Come già detto, le **istruzioni** date, hanno spesso il fine di manipolare delle informazioni contenute in variabili o in altre strutture dati. In questo capitolo vediamo quali sono le **operazioni** di base possibili su questi dati.

Iniziamo dalle semplici operazioni **aritmetiche**, perchè non necessitano di particolari spiegazioni: si effetuano con i convenzionali segni di somma (+), sottrazione (-), divisione(/) e moltiplicazione (*).Un'**operazione** su una variabile si può trascrivere usando due sintassi diverse.

Immaginiamo di voler aggiungere un valore noto (ad es.:5) ad una variabile 'X'. Potremo scrivere la stessa operazione in due modi diversi:

```
{
//metodo 1
X = X + 5;
//metodo 2
X += 5;
}
```

I due sistemi si possono usare con tutte le operazioni, siano esse aritmetiche o logiche. Per completezza riporto la lista completa di tutti gli operatori diponibili in GML, ripetendo anche quelli già citati. Nell'elenco sono separati da virgole ed i gruppi finiscono con duepunti, ma virgole e duepunti non sono operatori:

- **+, -, /, *:** sono gli operatori di somma, sottrazione, divisione e moltiplicazione.
- **div**, **mod**: divisione intera e modulo (restituisce il resto di una divisione intera).
- **not**, **and**, **or**, **xor**: operatori logici, rappresentabili coi simboli **!** (not), **&&** (and), **||** (or) e **^^** (xor).
- **~, &, |, ^, <<, >>:** complemento, and, or, xor, shift left, shift right... sono **operatori binari**.
- **<, <=, ==, !=, >=, >:** minore, minore o uguale, uguale, diverso, maggiore o uguale, maggiore sono gli operatori di comparazione disponibili.

Controllo del programma

Vediamo un **esempio** pratico d'impiego: costruiamo un'espressione per comandare una struttura di controllo. Ci serviremo dell'immaginazione e delle nozioni apprese fin ora.

Immaginiamo una **room** che contenga un oggetto in moto verso sinistra. Vogliamo che, uscendo dallo schermo, rientri dal lato opposto della stanza. La stanza è larga 640 **pixel**. Il bordo sinistro corrisponde a 0 pixel.

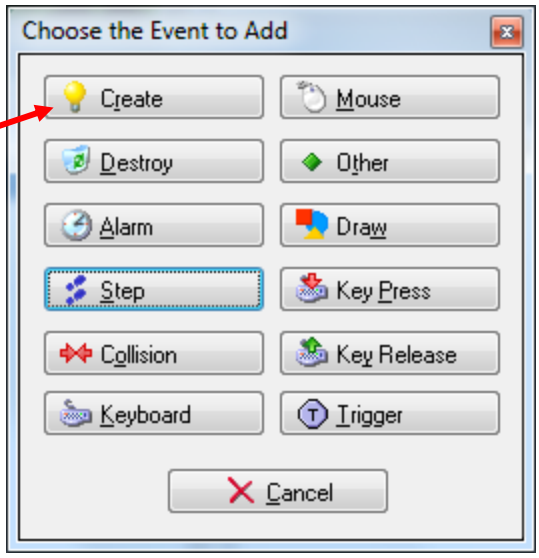
```
{
controllo=(x<0);

  if (controllo)
    x=640;
}
```

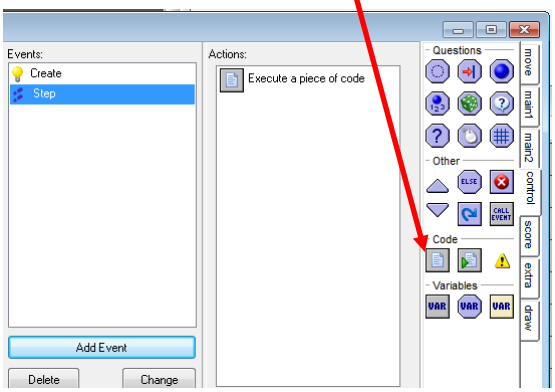
Vediamo come funziona: per prima cosa, con **var** si dichiara la variabile **CONTROLLO**. Questa sarà valida solo all'interno di questo blocco. A **CONTROLLO** viene assegnato il risultato di una **comparazione** tra la variabile **x**, che contiene la posizione dell'oggetto sullo schermo, e la posizione 0 (il bordo sinistro della stanza).

Finchè l'**oggetto** è lontano dal bordo dello schermo, la comparazione da come risultato 0. Quando **x** è uguale o minore di 0, **la comparazione** diventa 1. Viene eseguita l'istruzione x=640: l'oggetto va sul lato opposto della **room**. Vediamo di mettere l'esempio in pratica: **Esempio1**

- Aggiungete uno sprite: non importa cosa rappresenti, un quadratino nero andrà benissimo.
- Aggiungete un oggetto ed abbinategli lo sprite.
- Aggiungete all'oggetto l'evento **create**.



- Nella finestra action corrispondente all'evento create trascinate l'icona 'Execute a piece of code'.



- Si aprirà l'editor degli script. Scrivete:

```
x=320;
speed=-4;
direction=180;
```

- Aggiungete l'evento **step** e con la stessa procedura inserite il codice di prima:

```
{
    controllo=(x<0);

            if (controllo)
              x=640;
    }
```

- Aggiungete una **room** e mettete l'oggetto grossomodo al centro.
- Avviate il gioco vedrete lo sprite muoversi verso il bordo sinistro e quando lo raggiunge sembra rientrare da destra.

Le funzioni fornite da GML

Con le sole operazioni che abbiamo visto fin ora non saremmo in grado di realizzare **programmi** complessi, se non scrivendo chilometri di codice per compiere anche le operazioni più banali. Fortunatamente, **GML** ci fornisce molte **funzioni** che rispondono alle necessità più frequenti.

Una **funzione** è un pezzo di programma, scritto in modo da poter essere riutilizzato, specializzato nella soluzione di uno specifico problema.

In **GML** ci sono più di 1000 funzioni predefinite (1178 in GM6, 1233 in GM7), sufficienti a **risolvere** la quasi totalità dei nostri problemi. Se queste non fossero sufficienti, esistono **sistemi** per estendere le capacità del lingaggio.

Usare una funzione (si dice '**chiamare**' una funzione), equivale a scrivere una singola istruzione.

Alcune funzioni **restituiscono** dei valori al termine della loro esecuzione, altre eseguono solo delle procedure.

Alcune **funzioni** richiedono che gli vengano fornite delle informazioni su cui lavorare. Vediamo come.
La chiamata ad una funzione si esegue scrivendo il nome della stessa, seguito da parentesi tonde. Se la funzione richiede delle informazioni, queste vanno scritte tra le parentesi e prendono il nome di **argomenti** della funzione. Se la funzione richiede più argomenti, questi saranno separati da una virgola.

Vi chiederete com'è possibile **ricordarsi** il nome e gli argomenti attesi da oltre mille funzioni. Molto semplice: questo **non** è necessario. Primo di tutto, quelle usate frequentemente sono solo qualche decina.

In secondo luogo, nella parte bassa dello **script editor** c'è un aiuto in linea: non appena si scrivono due carattari, inizierà a mostrare in quest'area tutte le funzioni che iniziano con quelle lettere, indicando quali e quanti argomenti attendono.

Infine, le funzioni sono classificate per scopo: tutte quelle che riguardano l'uso della tastiera iniziano per **keyboard_**, quelle per la manipolazione delle stringhe iniziano per **string_** etc.

Vediamo un esempio di chiamata a funzione:

```
{
  if keyboard_check(vk_up)
{
  //blocco di istruzioni
  }
}
```

Come spiegato nei capitoli precedenti, il **blocco** di istruzioni dopo **if** viene eseguita se la condizione è **vera**.In questo caso, la **condizione** è il risultato di una chiamata a funzione: la funzione chiamata è **keyboard_check**(), che verifica se un tasto è premuto.

Nell'esempio il tasto controllato è rappresentato da **vk_up**, una costante che contiene il codice carattere della freccia verso l'alto, che viene passata alla funzione come argomento. Quando il tasto è premuto, la funzione restituisce 1 ed il blocco viene eseguito, altrimenti il flusso del programma prosegue senza fare nulla.

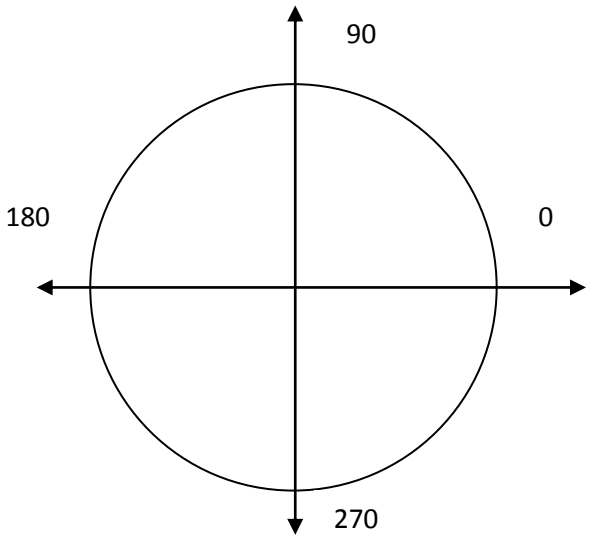
Come esempio supponiamo di controllare lo sprite di prima con le quattro frecce il codice che scriveremmo come action per l'evento step sarebbe: **Esempio2**

```
{
 if keyboard_check(vk_up){
  y-=5;
  direction=90;
 }

 if keyboard_check(vk_down){
  y+=5;
  direction=270;
 }

 if keyboard_check(vk_left){
  x-=5;
  direction=180;
 }

 if keyboard_check(vk_right){
  x+=5;
  direction=0;
 }
}
```



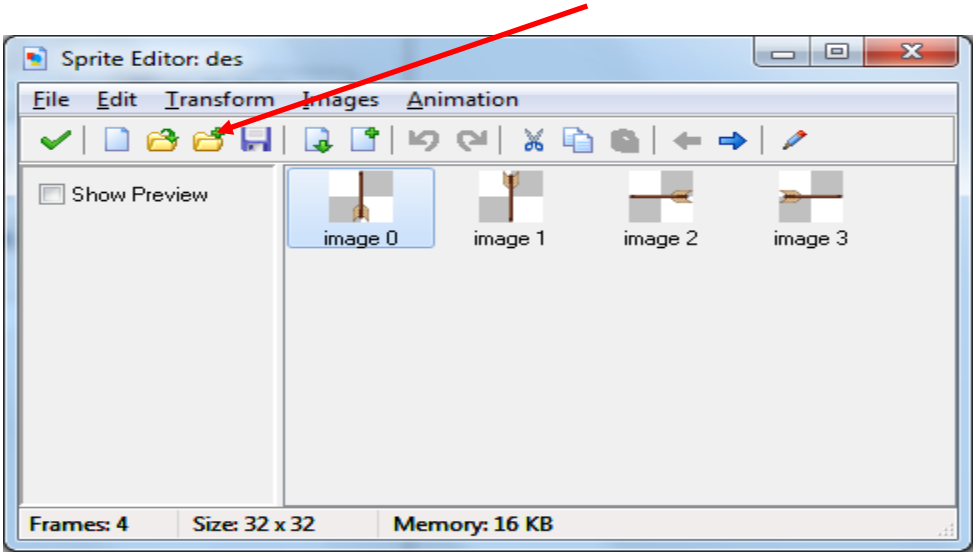
Come si vede dalla figura le direzioni sono espresse in gradi

Ricordiamoci di cancellare dallo script azione in risposta alla creazione l'istruzione: speed=4 altrimenti l'oggetto si muoverebbe comunque da solo.

Gli script possono anche essere scritti come script separati cliccando sul bottone create script o dal menu risorse voce create script. In questo caso è come se creassimo delle nuove funzioni scritte da noi e a come tutte le funzioni possiamo passare dei parametri. Nel corpo della funzione si accede ad essi mediante delle variabili predefinite chiamate argument0, arument1… ecc.

Come esempio supponiamo di creare un nuovo gioco nel quale creiamo una sprite freccia con 4 sottoimmagini. Dopo aver caricata la prima immagine freccia dello sprite si clicca su edit sprite e

dalla finestra seguente si aggiungono le altre immagini cliccando:



Adesso creiamo uno script con il comando crea script e inseriamo il seguente codice:

```
/*  Direzione di movimento e sprite usato

 argument0: direzione (1=SU, 2=GIU, 3=SX, 4=DX);
 */
 switch (argument0)
 {
  case 1: y-=5;
        direction=90;
        break;
  case 2: y+=5;
        direction=270;
        break;
  case 3: x-=5;
        direction=180;
        break;
  case 4: x+=5;
        direction=0;
        break;
 }
 image_index = argument0-1;
```

nel codice di azione per l'evento step inseriamo il seguente codice:

```
if keyboard_check(vk_up) MuoviFreccia(1);

if keyboard_check(vk_down) MuoviFreccia(2);

if keyboard_check(vk_left) MuoviFreccia(3);

if keyboard_check(vk_right) MuoviFreccia(4);
```

e ricordiamoci di modificare l'azione dell'evento create dell'oggetto freccia inserendo la riga di codice:

```
image_speed=0;
```

per fare in modo che non venga eseguita l'animazione sulle quattro frame dello sprite freccia.

Se tutto ha funzionato dovremmo avere la freccia che si muove nelle direzioni dei tasti cambiando immagine in base alla direzione. **Esempio3**

Abbiamo finora usato alcune delle variabili predefinite per gli oggetti in game maker vediamo ora nel dettaglio alcune delle più importanti:

x	**La coordinate x dell'istanza**
y	**La coordinate y dell'istanza**
hspeed	**La velocità orizzontale in pixel per ogni passo**
vspeed	**La velocità vrticale in pixel per ogni passo**
direction	**La direzione corrente del movimento espresso in gradi 0-360**
speed	**La velocità corrente nella direzione corrente**
visible	**L'oggetto è visibile valore 1 o invisibile valore 0**
image_index	**Questa variabile indica quale sottoimmagine dello sprite è attualmente mostrata. Una volta modificata può restare fissa se image_speed =0**
image_speed.	**Questa vriabile indica la velocità di animazione dello sprite di solito è impostata ad 1 se viene impostata a 0 lo sprite mostra una immagine fissa**
score	**Il valore corrente del punteggio**
lives	**Numero di vite**
health	**Stato di salute (0-100)**
mouse_x	**Coordinate x del mouse**
mouse_y	**Coordinata y del mouse**

Queste variabili esistono per ogni istanza degli oggetti utilizzati. Possono essere utilizzate da sole nello script dell'oggetto a cui si fa riferimento oppure con la notazione puntata per riferirsi ad un altro oggetto. Esempio altro.x rappresenta la coordinata x di un ipotetico altro oggetto.

Come abbiamo visto negli esempi precedenti ogni oggetto ha una sprite associato. Questa è una singola immagine o consiste di più immagini. Per ogni istanza dell'oggetto il programma disegna l'immagine corrispondente sullo schermo, con la sua origine (come definito nelle proprietà sprite) nella posizione (x, y) dell'istanza. Quando ci sono più immagini, cicla attraverso esse per ottenere un effetto di animazione. Ci sono un certo numero di variabili che influenzano il modo in cui l'immagine viene disegnata. Questi possono essere usati per cambiare gli effetti. Ogni istanza ha le seguenti variabili:

visible	Se visibile è true (1) l'immagine viene disegnata, altrimenti non viene disegnato. Istanze invisibili sono ancora attive e creno eventi di collisione; solo non li vedi. Impostare la visibilità su false è utile per esempio per oggetti controller (rendendoli non solidi per evitare collisioni) o interruttori nascosti.
sprite_index	Questo è l'indice dello sprite corrente per l'istanza. È possibile modificarlo per dare all'istanza uno sprite diverso. Come valore è possibile utilizzare i nomi dei diversi sprite definiti. Cambiare lo sprite non modifica l'indice del subimage attualmente visibile.
sprite_width*	Indica la larghezza dello sprite. Questo valore non può essere modificato, ma si può utilizzare.
sprite_height*	Indica l'altezza dello sprite. Questo valore non può essere modificato, ma si può utilizzare.
sprite_xoffset*	Indica l'offset orizzontale dello sprite, come definito nelle proprietà sprite. Questo valore non può essere modificato, ma si può utilizzare.
sprite_yoffset*	Indica l'offset verticale dello sprite, come definito nelle proprietà sprite. Questo valore non può essere modificato, ma si può utilizzare.
image_number*	Il numero di sottoimmagini per lo sprite corrente per l'istanza (non può essere modificato).
image_index	Se l'immagine ha più sottoimmagini il programma cicla attraverso di loro. Questa variabile indica la sootoiimagine corrente (sono numerate a partire da 0). È possibile modificare l'immagine corrente modificando questa variabile. Il programma continuerà a ciclare, a partire da questo nuovo indice.

image_speed	La velocità con la quale si passa da una sottoimmagine ad un'altra. Un valore pari a 1 indica che ogni passo si ottiene l'immagine successiva.Impostando a 0 questo valore si annulla l'animazione e si ha una immagine fissa
depth	Normalmente le immagini vengono disegnate nell'ordine in cui vengono create le istanze. È possibile modificare questo impostando la profondità dell'immagine. Il valore di default è 0, a meno che non si imposta un valore diverso nelle proprietà dell'oggetto. Più alto è il valore più l'istanza è lontana. Le istanze con la profondità maggiore si trovano dietro le istanze con una profondità inferiore
image_xscale	Un fattore di scala per rendere le immagini più grandi o più piccoli. Un valore pari a 1 indica la dimensione normale. È necessario impostare separatamente la XScale orizzontale e verticale yscale. La modifica della scala cambia anche i valori per la larghezza e l'altezza ed influenza gli eventi di collisione, come ci si potrebbe aspettare. La modifica della scala può essere utilizzato per ottenere un effetto 3-D. È possibile utilizzare un valore di -1 per riflettere lo sprite in senso orizzontale.
image_yscale	Fattore di scala verticale. Normalmente 1 è non ridimensionabile. È possibile utilizzare un valore di -1 per capovolgere lo sprite in senso verticale.